Selective Preemption Strategies for Parallel Job Scheduling*

Rajkumar Kettimuthu Vijay Subramani Srividya Srinivasan Thiagaraja Gopalasamy

D. K. Panda P. Sadayappan

Department of Computer and Information Science
The Ohio State University
$\{$*kettimut,subraman,srinivas,gopalsam,panda,saday*$\}$*@cis.ohio-state.edu*

Abstract

Although theoretical results have been established regarding the utility of pre-emptive scheduling in reducing average job turn-around time, job suspension/restart is not much used in practice at supercomputer centers for parallel job scheduling. A number of questions remain unanswered regarding the practical utility of pre-emptive scheduling. We explore this issue through a simulation-based study, using job logs from a supercomputer center. We develop a tunable selective-suspension strategy, and demonstrate its effectiveness. We also present new insights into the effect of pre-emptive scheduling on different job classes and address the impact of suspensions on worst-case slowdown.

1 Introduction

Although theoretical results on the effect of pre-emptive scheduling strategies in reducing average job turn-around time have been well established, pre-emptive scheduling is not currently being used for scheduling parallel jobs at supercomputer centers. Compared to the large number of studies that have investigated non-preemptive scheduling of parallel jobs, little research has been reported on empirical evaluation of preemptive scheduling strategies using real job logs [1, 2, 10, 7] .

The basic idea behind preemptive scheduling is simple: if a long running job is temporarily suspended and a waiting short job is allowed to run to completion first, the wait time of the short job is significantly decreased, without much fractional increase in the turn-around time of the long job. Consider a long job with runtime T_l. If after time t, a short job arrives with runtime T_s. If the short job were run after completion of the long job, the average job turnaround time would be $\frac{(T_l+(T_l+T_s-t))}{2}$, or $T_l + \frac{(T_s-t)}{2}$. Instead, if the long job were suspended when the short job arrived, the turnaround times of the short and long jobs would be T_s and $(T_s + T_l)$ respectively, giving an average of $T_s + \frac{T_l}{2}$. The average turnaround time with suspension is less if $T_s < T_l - t$, i.e. the remaining runtime of the running job is greater than the runtime of the waiting job.

*Supported in part by a grant from Sandia Laboratories

However, the use of a suspension criterion based simply on comparison of the remaining runtimes of jobs might result in starvation. It is desirable that the suspension strategy bring down the average slowdown without increasing the worst case slowdowns. Even though theoretical results have established that preemption improves the average turnaround time, it is important to perform evaluations of preemptive scheduling schemes using realistic job mixes derived from actual job logs from supercomputer centers, to understand the effect of suspension on various categories of jobs. The primary contributions of this paper are:

- The development of a selective-suspension strategy for pre-emptive scheduling of parallel jobs
- Characterization of the significant variability in the average job slowdown for different job categories
- The study of the impact of suspension on the worst case slowdowns of various categories and development of a tunable scheme to improve worst case slowdowns.

We study the effect of preemption on the performance of various categories of jobs using a locally developed backfill scheduler. The rest of the paper is organized follows. Section 2 presents some basic background on scheduling of parallel jobs. Section 3 discusses the workload characterization. In Section 4, a basic preemptive scheduling scheme is proposed and evaluated. In Section 5, the effect of the preemption scheme on the worst case slowdowns is analyzed and a tunable scheme is proposed and evaluated. In Section 6, the proposed preemption schemes are evaluated under conditions where user estimates of runtime are inaccurate. In Section 7, we evaluate the impact of job-suspension overheads on pre-emptive scheduling. Section 8 presents our conclusions.

2 Background and Related Work

Scheduling is usually viewed in terms of a 2D chart with time along one axis and the number of processors along the other. Each job can be thought of as a rectangle whose height is the user estimated run time and width is the number of processors required. The simplest way to schedule jobs is to use the First-Come-First-Served (FCFS) policy. This approach suffers from low system utilization. Backfilling [8, 9] was proposed to improve the system utilization and has been implemented in most production schedulers [6, 13]. Backfilling works by identifying "holes" in the 2D chart and moving forward smaller jobs that fit those holes. There are two common variations to backfilling - conservative and aggressive. In conservative backfilling, a smaller job is moved forward in the queue as long as it does not delay any previously queued job. In aggressive backfilling, a small job is allowed to leap forward as long as it does not delay the job at the head of the queue.

Some of the common metrics used to evaluate the performance of scheduling schemes are the average turnaround time and the average bounded slowdown. We use the bounded slowdown for our studies. The bounded slowdown of a job is defined as follows:

$$\text{Bounded Slowdown} = \frac{(Waittime + Max(Runtime, 10))}{Max(Runtime, 10)}$$

The threshold of 10 seconds is used to limit the influence of very short jobs on the metric.

Pre-emptive scheduling aims at providing lower delay to short jobs relative to long jobs. Since long jobs have greater tolerance to delays as compared to short jobs, our suspension criterion is based on the eXpansion Factor (XFactor), which increases rapidly for short jobs and gradually for long jobs.

$$\text{XFactor} = \frac{(Waittime + EstimatedRunTime)}{EstimatedRunTime}$$

Although pre-emptive scheduling is universally used at the operating system level to multiplex processes on single-processor systems and shared-memory multi-processors, it is rarely used in parallel job scheduling. A large number of studies have addressed the problem of parallel job scheduling (see [5] for a survey of work on this topic), but most of them address non-preemptive scheduling strategies. Further, most of the work on pre-emptive scheduling of parallel jobs considers the jobs to be malleable [3, 10, 12, 14], i.e. the number of processors used to execute the job is permitted to vary dynamically over time.

In practice, parallel jobs submitted to supercomputer centers are generally rigid, i.e. the number of processors used to execute a job is fixed. Under this scenario, the various schemes proposed for a malleable job model are inapplicable. We address pre-emptive scheduling under a model of rigid jobs, where the pre-emption is “local”, i.e. the suspended job must be re-started on exactly the same set of processors on which they were suspended.

In a recent study [2], a pre-emptive scheduling strategy called the "Immediate Service (IS)" scheme was evaluated for shared-memory systems. With this scheme, each arriving job was given an immediate time-slice of 10 minutes, by suspending one or more running jobs if needed. The selection of jobs for suspension was based on their instantaneous-XFactor, defined as (wait time + total accumulated run time)/ (total accumulated run time). Jobs with the lowest instantaneous-XFactor were suspended. The IS strategy was shown to significantly decrease the average job slowdown for the traces simulated. A potential shortcoming of the IS scheme is that its preemption decisions are not in any way reflective of the expected runtime of a job. The IS scheme can be expected to provide significant improvement to the slowdown of aborted jobs in the trace. So it is unclear how much, if any, of the improvement in slowdown was experienced by the jobs that completed normally - however, no information was provided on how different job categories were affected. Chiang et al [1] examine the run-to-completion policy with a suspension policy that allows a job to be suspended at most once. Both these approaches limit the number of suspensions while we use a more selective approach to control the rate of suspensions, without limiting the number of times a job can be suspended. In [10], the design and implementation of a number of multiprocessor preemptive scheduling disciplines are discussed. They study the effect of preemption under the models of rigid, migratable and malleable jobs. They conclude that the preemption scheme that they propose may increase the response time for the model of rigid jobs.

So far, very few simulation based studies have been done on preemption strategies for clusters. If process migration is not allowed (due to the significant practical complications it entails), preemptive scheduling on distributed memory systems imposes an additional constraint that the suspended jobs should be restarted on the same set of physical processors. In this paper we propose tunable suspension strategies for parallel job scheduling in environments where process migration is not feasible.

3 Workload Characterization

From the collection of workload logs available from Feitelson’s archive [4], the CTC workload trace was used to evaluate the proposed schemes. This trace was generated by a 430 processors system. In order to reduce the time taken to run the set of simulations, a contiguous 5000 job subset of the trace was used (corresponding to roughly one month’s jobs).

Under normal load, with the standard non-preemptive aggressive backfilling strategy, using FCFS as the scheduling priority, the utilization was 51 percent. Although it is known that user estimates are quite inaccurate in practice, as explained above, we first studied the effect of preemptive scheduling under the idealized assumption of exact estimation, before studying the effect of inaccuracies in user estimates of job run time. Also, we first studied the impact of pre-emption under the assumption that the overhead for the suspension and restart is negligible and then studied the influence of the overhead.

3.1 Job Classification

Any analysis that is based on the aggregate slowdown of the system as a whole does not provide insights into the variability within different job categories. Therefore in our discussion, we classify the jobs into various categories based on the runtime and the number of processors requested, and analyze the slowdown for each category.

To analyze the performance of jobs of different sizes and lengths, jobs were classified into 16 categories: four categories based on their run time - Very Short(VS), Short(S), Long(L) and Very Long(VL) and four categories based on the number of processors requested - Sequential(Seq), Narrow(N), Wide(W) and Very Wide(VW). The criteria used for job classification are shown in Table 1. Table 2 shows the percentage of jobs in the trace, corresponding to the sixteen categories.

Job Categorization Criteria

	1 Proc	2-8Procs	9-32Procs	>32 Procs
0-10min	VS Seq	VS N	VS W	VS VW
10min-1hr	S Seq	S N	S W	S VW
1hr-8hr	L Seq	L N	L W	L VW
>8hr	VL Seq	VL N	VL W	VL VW

Table 1. Categorization of jobs based on their Runtime and Width.

Job Percentage

	1 Proc	2-8 Procs	9-32 Procs	>32 Procs
0-10min	14	8	13	9
10min-1hr	18	4	6	2
1hr-8hr	6	3	9	2
>8hr	2	2	1	1

Table 2. Category based Job Distribution.

Average Slowdown for Non Preemptive Scheduling

	1 Proc	2-8 Procs	9-32 Procs	>32 Procs
0-10min	2.6	4.76	13.01	34.07
10min-1hr	1.26	1.76	3.04	7.14
1hr-8hr	1.13	1.43	1.88	1.63
>8hr	1.03	1.05	1.09	1.15

Table 3. Average slowdown for various categories with non preemptive scheduling.

Table 3 shows the average slowdowns for the different job categories under a non-preemptive aggressive backfilling strategy. The Overall slowdown was 3.58. Even though the overall slowdown is low, from the table it can be observed that one of the Very Short categories has average slowdown as high as 34. Preemptive strategies can be effective in reducing the high average slowdowns for the short categories, without significant degradation for long jobs.

4 Selective Suspension

We first propose a preemptive scheduling scheme, called the Selective Suspension (SS) scheme, where an idle job can preempt a running job if its priority is sufficiently higher than the running job. An idle job attempts to suspend a collection of running jobs so as to obtain enough free processors. In order to control the rate of suspensions, a suspension factor (SF) is used. This specifies the minimum ratio of the suspension threshold of a candidate idle job to the suspension threshold of a running job for preemption to occur. The suspension threshold used is the XFactor of the job.

4.1 Theoretical Analysis

Let T_1 and T_2 be two tasks submitted to the scheduler at the same time. Both tasks are of same length and require the entire system for execution and the system is free when the two tasks are submitted. Let 's' be the suspension factor. Before starting, both tasks have a suspension threshold of 1. The suspension threshold of a task remains constant when the task executes and increases when the task waits. One of the two tasks, say T_1, will be started instantaneously. The other task, say T_2, waits until its suspension threshold τ_2 becomes 's' times the threshold of T_1 before it can preempt T_1. Now T_1 waits until its suspension threshold τ_1 becomes 's' times τ_2 before it can preempt T_2. This occurs repeatedly. The optimal value for SF to restrict the number of repeated suspensions by two similar tasks arriving at the same time can be obtained as follows:

Let τ_w represent the suspension threshold of the waiting job and τ_r represent the suspension threshold of the running job.

Condition for the first suspension: τ_w = s

The preemption swaps the running job and the waiting job. So after the preemption, τ_w = 1 and τ_r = s.

Condition for second suspension: $\tau_w = s * \tau_r$ i.e. $\tau_w = s^2$

Similarly, the condition for n^{th} suspension is $\tau_w = s^n$.

The lowest value of s for which at most n suspensions occur is given by,

$\tau_w = s^{n+1}$, when the running job completes.

When the running job completes,

$\tau_w = \frac{waittime+runtime}{runtime}$ i.e $\tau_w = 2$; since the wait time of the waiting job equals the run time of the running job

$s^{n+1} = 2$, i.e. $s = 2^{\frac{1}{n+1}}$

Thus, if the number of suspensions has to be 0, then s = 2. For at most 1 suspension, we get s as $\sqrt{2}$. With s=1, the number of suspensions is very large, only bounded by the granularity of the preemption routine. With all jobs having equal length, any suspension factor greater than 2 will not result in any suspension and will be same as the suspension factor 2. However, with jobs of varying length, the number of suspensions reduces with higher suspension factors. Thus, in order to avoid thrashing and to reduce the number of suspensions, we use different suspension factors between 1.5 and 5 in evaluating our schemes.

4.2 Preventing Starvation without Reservation Guarantees

An idle job can preempt a running job only if its threshold is at least SF times greater than the threshold of the running job. All the idle jobs that are able to find the required number of processors by suspending lower threshold running jobs are selected for execution by preempting the corresponding jobs. All backfill scheduling schemes use job reservations for one or more jobs at the head of the idle queue as a means of guaranteeing finite progress, thereby assuring freedom from starvation. But the start time guarantees do not make much sense in the presence of preemption. Even if start time guarantees are given to the jobs in the idle queue, they are not guaranteed to run to completion since they can be suspended. However, because the SS strategy uses the expected slowdown as the suspension threshold, there is an automatic guarantee of freedom from starvation - ultimately any job's expected slowdown factor will get large enough that it will be able to preempt some running job and begin execution. Therefore, it is possible to run the backfill algorithm without the usual reservation guarantees. So, we remove the guarantees for all our preemption schemes, since the absence of reservations in the schedule facilitates better backfilling..

Further, jobs in some categories inherently have a higher probability of waiting longer in the queue than a job with comparable XFactor from another job category. For example, consider a VW job needing 300 processors, and a Seq job in the queue at the same time. If both jobs have the same XFactor, the probability that the Seq finds a running job to suspend is higher than the probability that the VW job finds enough lower threshold running jobs to suspend. Therefore, the average slowdown of the VW category will tend to be higher than the Seq category. To redress this inequity, we impose a restriction that the number of processors requested by a suspending job should be at least half of the number of nodes requested by the job that it suspends, thereby preventing the wide jobs from being suspended by the narrow jobs. The scheduler periodically (after every minute) invokes the preemption routine.

4.3 Algorithm

Let τ_i be suspension threshold for a task T_i which requests n_i processors. Let N_i represent the set of processors allocated to T_i. Let F_t represent the set of free processors and f_t represent the number of free processors at time 't' when the preemption is attempted.

Let candidates(T_i) represent the set of tasks that can be preempted by task T_i.

candidates(T_i) = { T_j : $\tau_i >$ SF * τ_j and $\frac{n_j}{n_i} < 2$ }

T_i can be scheduled by preempting one or more tasks in candidates(T_i) if and only if

$n_i \leq (\Sigma n_j + f_t) \, \forall T_j \, \epsilon$ candidates(T_i)

If T_i is itself a previously suspended task attempting reentry, the processor restriction also applies. So the above condition becomes:

candidates(T_i) = { T_j:τ_i>SF*τ_j and $\frac{n_j}{n_i}$<2 and $N_i \cap N_j \neq \emptyset$}

T_i can be scheduled by preempting one or more tasks in candidates(T_i) if and only if

$n_i \leq (\Sigma n_j + f_t) \, \forall T_j \, \epsilon$ candidates(T_i)

and

$N_i \subseteq F_t \cup \; N_j \, \forall T_j \, \epsilon$ candidates(T_i)

For both of the above scenarios, T_i preempts tasks in candidates(T_i) as given by the following condition:

The set of tasks suspended by T_i is

P = { T_j:$T_j \epsilon$candidates(T_i) and $n_i \leq f_t + \Sigma n_j$ } and (f_t +($\Sigma n_k \, \forall T_k \, \epsilon$ P) - ($n_m : T_m \, \epsilon$ P)) $< n_i$

In essence, the algorithm sorts the list of running jobs in ascending order of the suspension threshold and the list of idle jobs in descending order of suspension threshold. Then for each idle job, a minimal set of running jobs which satisfy the following conditions are chosen for suspension.

- The number of processors used by the running job is less than twice the number of processors requested by the idle job.
- The suspension threshold of the idle job is atleast SF times the suspension threshold of the running job.
- The sum of processors of all the running jobs in the minimal set, together with the number of free processors in the system at that instant is greater than or equal to the number of processors requested by the idle job.

4.4 Results

We compare the SS scheme run under various suspension factors with the No-Suspension(NS) scheme with aggressive Backfilling and the IS scheme. From Figure 1, we can see that the SS scheme provides significant improvement for the Very-Short(VS) and Short(S) length categories and Wide(W) and Very-Wide(VW) width categories. For example, for the VS-VW category, slowdown is reduced from 34 for the NS scheme to under 3 for SS with SF=2. For VS and S length categories, lower SF results in lower slowdown. This is because a lower SF increases the probability that a job in these categories will suspend a job in the Long(L) or Very-Long(VL) category. The same is also true for the L length category, but the effect of change in SF is less pronounced. For the VL length category, there is an opposite trend with decreasing SF, i.e. the slowdown increases. This is due to the increasing probability that a Long job will be suspended by a job in a shorter category as SF decreases. In comparison to the base No-Suspension(NS) scheme, the SS scheme provides significant benefits for VS and S categories, a slight improvement for most of the Long categories, but is slightly worse for the VL categories.

The performance of the IS scheme is very good for the VS category. It is better than the SS scheme for the VS length category and worse for the other categories. Even though the overall slowdown for IS is considerably less than the No-Suspension scheme, it is not better than SS. Moreover, in IS the VW and VL categories get significantly worse.

5 Tunable Selective Suspension (TSS)

From the graphs of the previous section, it can be observed that the SS scheme significantly improves the average slowdown of various job categories. But from a practical point of view, the worst case slowdowns are very important. A scheme that improves the average case slowdowns for most of the categories, but makes the worst case slowdown for the long categories worse, is not a desirable scheme. For example, a delay of 1 hour for a 10 minute job (slowdown = 7) is tolerable whereas a slowdown of 7 for a 24 hour job is unacceptable.

In Figure 2, we compare the worst case slowdowns for SF=2 with the worst case slowdowns of the NS scheme and the IS scheme. It can be observed that the worst case slowdown for the SS scheme is much better than the NS scheme for most of the cases. But the worst case slowdown for some of the long categories is higher than the NS scheme. Even though the worst case slowdown for SS is less than that of NS, the worst case slowdowns are much higher than the corresponding actegory averages for some of the short categories. For the IS scheme, the worst case slowdown for the very short categories is lower but much higher for the long categories, which is highly undesirable. We next propose a tunable scheme to improve the worst case slowdowns without losing the improvement in the average slowdowns. This is done by controlling the variance

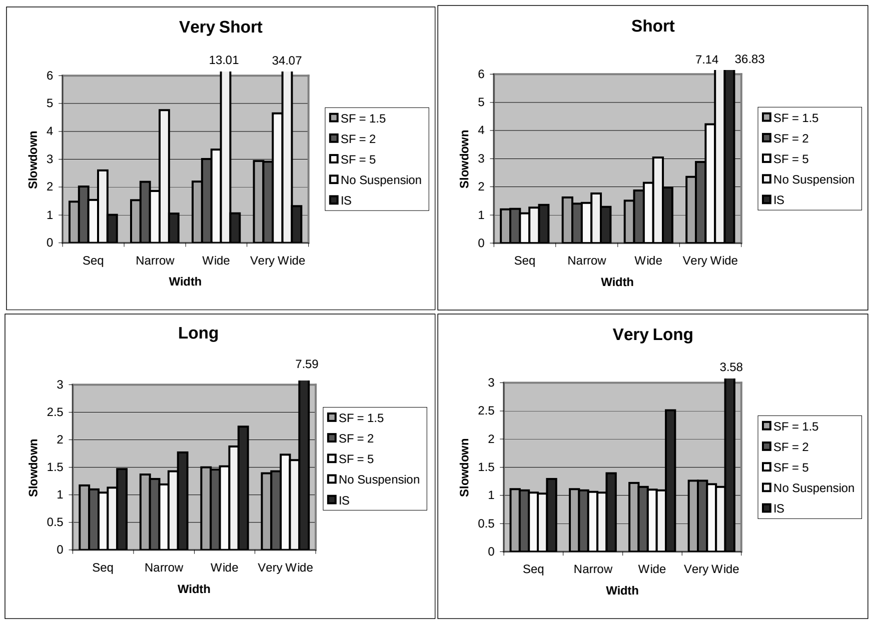

Figure 1. Comparison of the average slowdown of the SS Scheme with the NS and IS Schemes. Compared to NS, SS provides significant benefit for the VS, S, W and VW categories, slight improvement for most of L categories, but a slight deterioration for the VL categories. Compared to IS, SS performs better for all the categories except for the VS categories.

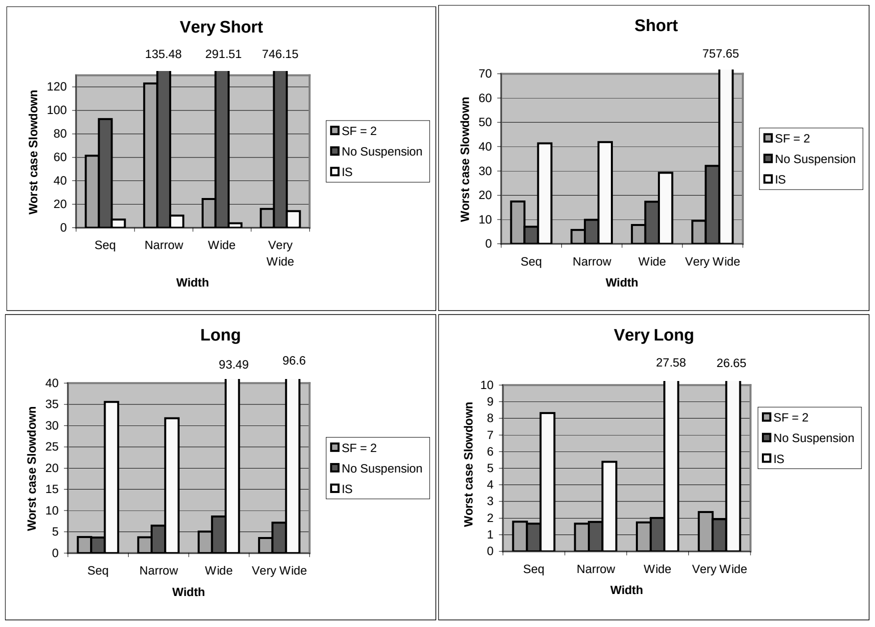

Figure 2. Comparison of the Worstcase Slowdowns of the SS Scheme with the NS and IS Schemes. SS is much better than NS for most of the categories and is slightly worse for some of the VL categories. Compared to IS, SS is much better for all the categories except for the VS categories.

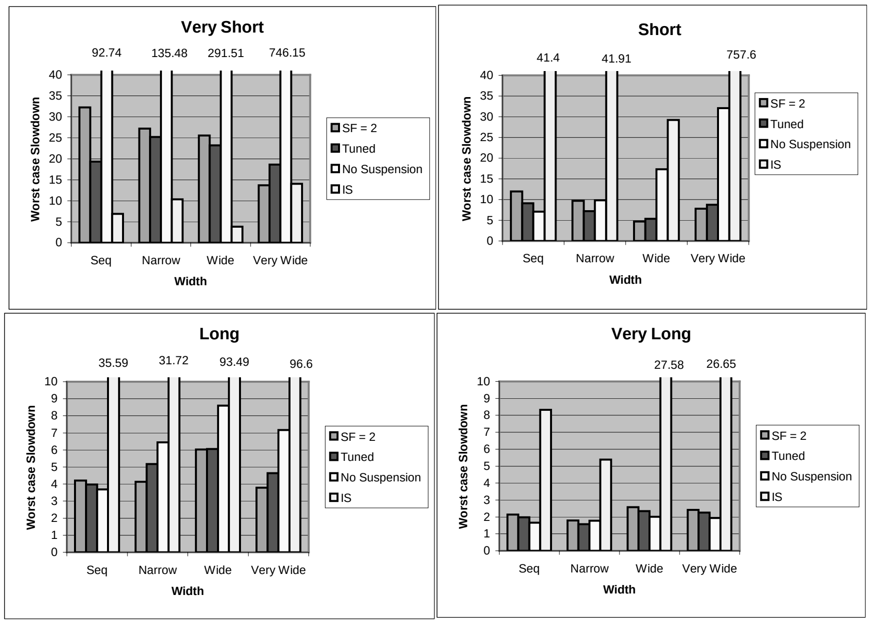

Figure 3. Comparison of the Worstcase Slowdowns of the TSS Scheme with SS, NS and IS Schemes. TSS improves the worstcase slowdown for the VL categories and some of the S categories without adversely affecting the other categories.

in the slowdowns by associating a limit with each job. Preemption of a job is disabled when its threshold exceeds this limit. This limit is set to 1.5 times the average slowdown of the category that the job belongs to.

5.1 Control of Variance

Task T_i preempts tasks in candidates(T_i) as given by the following condition:

The set of tasks suspended by T_i is
P = { T_j:$T_j \epsilon$candidates(T_i) and $n_i \le f_t + \Sigma n_j$ and $\tau_j \le 1.5$ $*SD_{avg}$(category(T_j))} and (f_t + (Σn_k $\forall T_k$ ϵ P) - (n_m : T_m ϵ P)) $< n_i$

where SD_{avg}(category(T_j)) represents the average slowdown in the SS scheme for the job category to which T_j belongs.

5.2 Results

Figure 3 shows the results for the tunable suspension scheme. It improves the worst case slowdowns for some long categories (VL W, VL VW, L N) and some short categories (VS Seq, VS N, S Seq) without affecting the worst case slowdowns of the other categories. This scheme can also be applied to selectively tune the slowdowns for particular categories.

6 Inaccurate User Estimates

We have so far assumed that the user estimates of job runtime are perfect. Now, we consider the effect of user estimate inaccuracy on the proposed schemes. This is desirable from the point of view of realistic modeling of an actual system workload. In practice, the scheduler only has information about the job wallclock limit specified by the user, and not the actual execution time. Hence it is important to carry out simulations where the scheduler bases its decisions on the user specified wallclock limit, which can often be quite inaccurate. Most simulation based studies of job scheduling have done this, but we believe that there is a problem that has not been recognized by previous studies. Abnormally aborted jobs tend to excessively skew the average slowdown of jobs in a workload. Consider a job requesting a wall-clock limit of 24 hours, that is queued for 1 hour, and then aborts within one minute due to some fatal exception. The slowdown of this job would be computed to be 60, whereas the average slowdown of normally completing long jobs is typically under 2. If even 5% of the jobs have a high slowdown of 60, while 95% of the normally completing jobs have a slowdown of 2, the average slowdown over all jobs would be around 5. Now consider a scheme such as the speculative backfilling strategy evaluated in [11]. With this scheme, a job is given a free timeslot to execute in, even if that slot is considerably smaller than the requested wall-clock limit. Aborting jobs will quickly terminate, and since they did not have to be queued till an adequately long window was available, their slowdown would decrease dramatically with the speculative backfilling scheme. As a result, the average slowdown of the entire trace would now be close to 2, assuming that the slowdown of the normally completing jobs does not change significantly. A comparison of the average slowdowns would seem to indicate that the speculative backfill scheme results in a significant improvement in job slowdown from 5 to 2. However, under the above scenario, the change is only due to the small fraction of aborted jobs, and not due to any benefits to the normal jobs. In order to avoid this problem, we group the jobs into two different estimation categories. The jobs that are well-estimated (the estimated time is not more than twice the actual runtime of that job) and badly estimated jobs (the estimated runtime

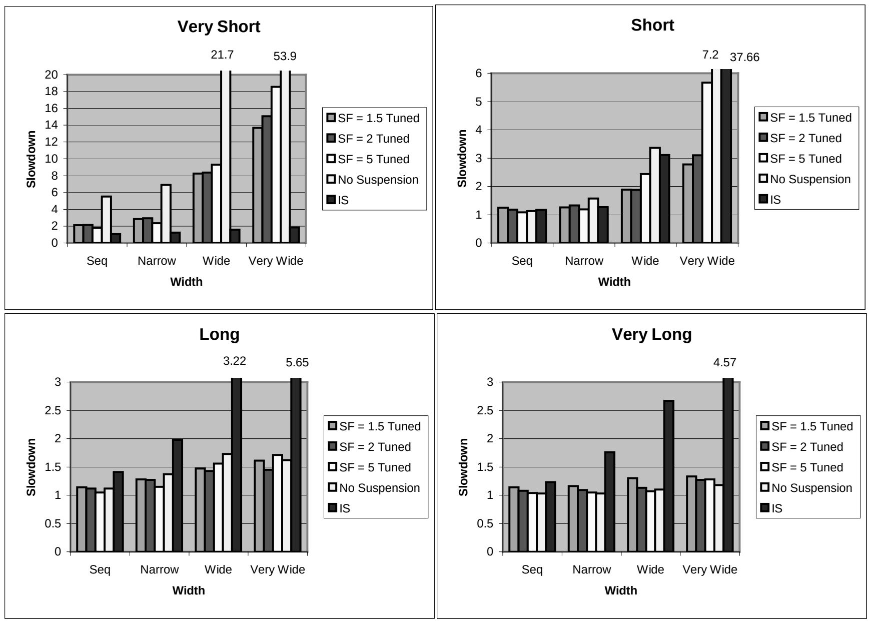

Figure 4. Comparison of the Average Slowdown of the TSS Scheme with NS and IS Schemes with inaccurate user esitmates of runtime. Similar trends as are observed here as for the case with accurate user estimates.

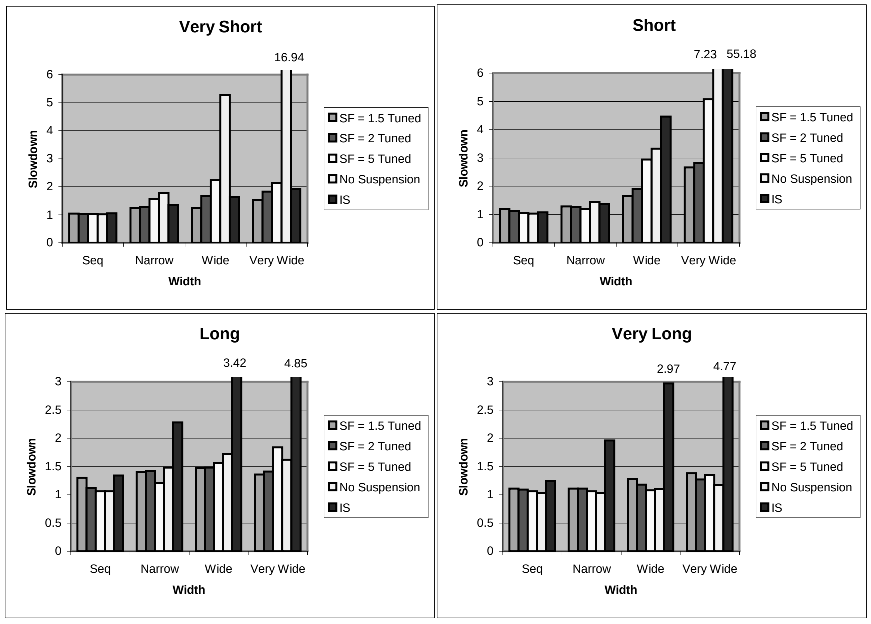

Figure 5. Comparison of the Average Slowdown of the TSS Scheme with NS and IS Schemes for the Well-estimated Jobs. The trends are similar to that of the accurate user estimate case for the S, L and VL categories. The performance of the SS scheme for the VS categories is better or comparable to that of the IS scheme.

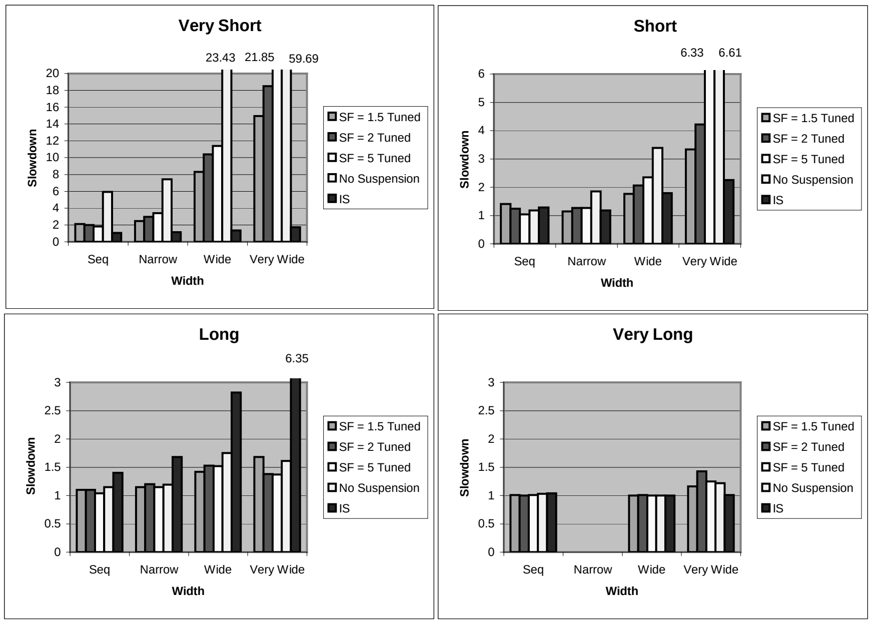

Figure 6. Comparison of the Average Slowdown of the TSS Scheme with NS and IS Schemes for the Poorly-estimated Jobs. The trends are similar to that of the accurate user estimate case for the S, L and VL categories. The SS scheme tends to penalize very poorly estimated jobs that belong to the VS category.

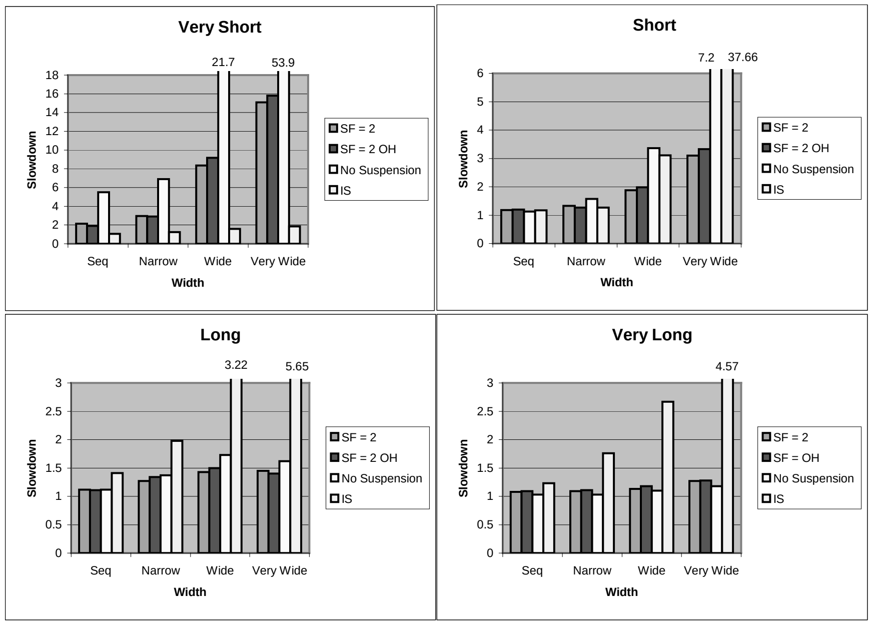

Figure 7. Comparison of the Average Slowdown of the TSS scheme with IS and NS schemes, with modeling of overhead for suspending/restarting a job. The performance of the TSS scheme with modeling of overhead is comparable to its performance in the absence of overhead.

is more than twice the actual runtime). Within each group, the jobs are further classified into the 16 categories based on their actual run time and the number of processors requested.

It can be observed from Figure 4 that the Selective Suspension Scheme improves the slowdowns for most of the categories without affecting the other categories. The slowdowns for the short and wide categories are quite high compared to the other categories and this is mainly because of the over estimation. Since the suspension threshold used by the SS scheme is xfactor, it favors the short jobs. But if a short job was badly estimated, it would be treated as a long job and its priority would increase only gradually. So it would not be able to suspend running jobs easily and therefore end up with a large slowdown. This does not happen with IS because of the 10 minute time quantum for each arriving job, irrespective of the estimated run time. Therefore, the slowdowns for the very short category (whose length is less than or equal to 10 minutes) are lower with IS than other schemes. However, for the other categories, SS performs much better than IS.

In Figures 5 and 6, the performance data for the various job categories is shown separately for the well estimated jobs and the poorly estimated jobs. It is evident that the higher slowdowns for the VS categories in SS is due to the poorly estimated jobs. It can also be observed that, for the well estimated jobs, SS is comparable to IS for the VS categories and SS outperforms IS for all other categories.

7 Modeling of Job Suspension Overhead

We have so far assumed no overhead for pre-emption of jobs. In this section, we report on simulation results that incorporate overheads for job suspension. Since the job trace did not have information about job memory requirements, we considered the memory requirement of jobs to be random and uniformly distributed between 100MB and 1GB. The overhead for suspension is calculated as the time taken to write the main memory used by the job to the disk. The memory transfer rate that we considered is based on the following scenario: With a commodity local disk for every node, with each node being a quad, the transfer rate per processor was assumed to be only 2 MBps.

Figure 7 compares the slowdowns of the proposed tunable scheme with NS and IS in the presence of overhead for the suspension/restart. It can be observed that overhead does not significantly affect the performance of the Tunable Suspension Scheme.

8 Conclusions

In this paper, we have explored the issue of pre-emptive scheduling of parallel jobs, using a job trace from a supercomputer center. We have proposed a tunable, selective suspension scheme and demonstrated that it provides significant improvement in the average slowdown and the worst case slowdowns of most job categories. It was also shown to provide better slowdown for most job categories over a previously proposed Immediate Service scheme. We also modeled the effect of overheads for job suspension, showing that the proposed scheme provides significant benefits over non-preemptive scheduling and the Immediate Service strategy. We also evaluated the proposed schemes in the presence of over estimations and showed that it produced good results.

Acknowledgments

We would like to thank the anonymous referees for their helpful suggestions on improving the presentation of the paper.

References

[1] S. H. Chiang, R. K. Mansharamani, and M. K. Vernon. Use of application characteristics and limited preemption for run-to-completion parallel processor scheduling policies. In *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 33–44, 1994.

[2] S. H. Chiang and M. K. Vernon. Production job scheduling for parallel shared memory systems. In *Proceedings of International Parallel and Distributed Processing Symposium*, 2002.

[3] X. Deng, N. Gu, T. Brecht, and K. Lu. Preemptive scheduling of parallel jobs on multiprocessors. In *SODA: ACM-SIAM Symposium on Discrete Algorithms*, 1996.

[4] D. G. Feitelson. Logs of real parallel workloads from production systems. URL: http://www.cs.huji.ac.il/labs/parallel/workload/logs.html.

[5] D. G. Feitelson, L. Rudolph, U. Schwiegelshohn, K. C. Sevcik, and P. Wong. Theory and practice in parallel job scheduling. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 1–34. Springer Verlag, 1997. Lect. Notes Comput. Sci. vol. 1291.

[6] D. Jackson, Q. Snell, and M. Clement. Core algorithms of the Maui scheduler. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 87–102. Springer Verlag, 2001. Lect. Notes Comput. Sci. vol. 2221.

[7] L. T. Leutenneger and M. K. Vernon. The performance of multiprogrammed multiprocessor scheduling policies. In *ACM SIGMETRICS Conference on Measurement and Modelling of Computer Systems*, pages 226–236, May 1990.

[8] D. Lifka. The ANL/IBM SP scheduling system. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 295–303. Springer-Verlag, 1995. Lect. Notes Comput. Sci. vol. 949.

[9] A. W. Mu'alem and D. G. Feitelson. Utilization, predictability, workloads, and user runtime estimates in scheduling the ibm sp2 with backfilling. In *IEEE Transactions on Parallel and Distributed Systems*, volume 12, pages 529–543, 2001.

[10] E. W. Parsons and K. C. Sevcik. Implementing multiprocessor scheduling disciplines. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 166–192. Springer Verlag, 1997. Lect. Notes Comput. Sci. vol. 1291.

[11] D. Perkovic and P. J. Keleher. Randomization, speculation, and adaptation in batch schedulers. *Cluster Computing*, 3(4):245–254, 2000.

[12] K. C. Sevcik. Application scheduling and processor allocation in multiprogrammed parallel processing systems. *Performance Evaluation*, 19(2-3):107–140, 1994.

[13] J. Skovira, W. Chan, H. Zhou, and D. Lifka. The EASY - LoadLeveler API project. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 41–47. Springer-Verlag, 1996. Lect. Notes Comput. Sci. vol. 1162.

[14] J. Zahorjan and C. McCann. Processor scheduling in shared memory multiprocessors. In *ACM SIGMETRICS Conference on Measurement and Modelling of Computer Systems*, pages 214–225, May 1990.